An Architectural Description Language for Enterprise Computing

Iman Poernomo
DSTC Pty Ltd
imanp@dstc.com

Heinz Schmidt
School of Computer Science and Software Engineering,
Monash University
hws@csse.monash.edu.au

Architectural description languages (ADLs) are used to specify high-level, compositional view of a software application. ADLs usually come equipped with a rigourous state-transition style semantics, facilitating specification and analysis of distributed and event-based systems.

However, enterprise system architectures built upon newer middleware (implementations of Java's EJB specification, or Microsoft's COM+/ .NET) require additional expressive power from an ADL. The TrustME *ADL is designed to meet this need. In this paper, we describe several aspects of* TrustME *which facilitate specification and anlysis of middleware-based architectures for the enterprise.*

1. Introduction

Over the past decade, middleware has undergone considerable evolution to meet the needs of the enterprise. The enterprise requires software solutions which are business-oriented, mission-critical, scalable and distributed. It is now generally accepted that such solutions can be delivered effectively by utilizing a component-based middleware that incorporates a range of enterprise services (such as security services or transactional services). Examples of such newer middleware are those based on Java's EJB specification or Microsoft's COM+/.NET.

The complex nature of enterprise requirements entails that solution providers will profit from some formal design methodology. There are several specification approaches which are applicable to various facets of enterprise development. For instance, UML [1] can be useful for providing a clear object-oriented design for components. Also, process modelling [2] allows the developer to represent and analyse business processes prior to implementation.

In this paper, we outline an approach to specification and analysis of architectures for middleware-based enterprise solutions. We take the definition of *software architecture* as given in [3, pp. 10–12]:

> Software architecture is a set of concepts and design decisions about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domain.

Architectural description languages (ADLs) present a compositional, component-oriented view of software achitecture. Most ADLs achieve this through modelling configurations of

- *components*, asynchronous points of computation in a system,
- *connections* between *ports* or *services* of these components, the possible types of communication and interaction within a system, and
- *compound components*, higher-level components composed of interconnected lower-level components [4, 5].

Because components are asynchronous, and connections are a form of loosely coupled communication, ADLs are well suited to describe concurrent, distributed and/or event-based software. Also, most ADLs have a well-defined behaviorial semantics, so that configurations have a meaning which can be analysed (to prove, for instance, safety and liveness properties). In this way, ADLs have a syntax and semantics aimed at defining and analysing distributed configurations of components.

Thus, current ADLs are ideal for describing distributed systems and for describing solutions based on older middleware, such as Microsoft's DCOM and basic CORBA implementations. This is true because, from an architectural perspective, there is little difference between a distributed, older middleware-based system and a concurrent, event-based system.

However, some challenges remain concerning architectural modelling for the enterprise. We identify four issues which arise when modelling systems based on newer, enterprise-scale middleware:

1. *Safer connections between components.* Most ADLs model a component's interface as a set of port or service names. Semantically, these correspond to actions or events which might occur through executing a component. Most ADLs define interface elements to possess no signature or specification, so a binding is simply a pair of two such ports. Thus, the task of determining whether ports should be connected is orthogonal to architectural design.

   This can be acceptable for defining small-scale architectures, where communication between components is simple (such as Unix scripts which use pipes and filters). Unfortunately, for larger architectures, where complex information is being communicated, this situation leads to increased risk of design failure. This situtation is therefore not satisfactory for scalable and mission-critical applications.

2. *Closer correspondence to middleware component models.* It is desirable that architecturally significant elements of a middleware application have accurate and recognizable representation within the ADL. This has several advantages for enterprise development. First, the ADL is made more intuitive, because software engineers can retain a unified understanding how the the ADL relates to the implementation. Second, it must be remembered that an ADL is not only intended to be a semantic model of computation for a system, but also to be a tool for syntactic description of compositional structure. If architecturally signficant elements are not present in an ADL, then the ADL does not satisfy this requirement (even if the ADL can model such elements by some intermediate translation).

   For instance, the "components" of an ADL should be recognizable as components by programmers famililar with middleware component models. For example, an ADL should incorporate notions such as multiple interfaces, subtyping and substitutability.

   Another architecturally significant feature common to newer middleware is *context-based interception* (in COM+) or *containers/servers* (in EJB). Here, deployed components are conceived as residing within a context (or container) that potentially intercepts and manipulates each call that crosses the context boundary. A range of *contextual services* are offered by the middleware. Examples of such services are security and transactional services. These services may be used by components in various ways, determined by *declarative deployment constraints* associated with component instances. Contexts provide a pre-programmed scalable, mission-critical infrastructure for housing components, enabling the developer to focus on business-oriented design and programming.

   Because context-based interception is a valuable feature of newer, enterprise-oriented middleware, it is important that an enterprise-oriented ADL can provide a direct representation of contexts and contextual services.

3. *Integration with other software models.* The architectural description of a system is not a standalone view. Other system views and design methodologies will be necessary to meet the requirements of the enterprise.

   Within the software industry, UML is now firmly established as the object-oriented specification standard. A UML object-oriented model provides an important lower-level view of an enterprise system. For instance, such model should be required in designing basic components of the system. The UML metamodel is based upon the MOF [6] metametamodel. The intention of the MOF is that designs of metamodels for various system views may be given within the same framework, to understand their inter-relationships. Besides UML, an example of such a MOF-based metamodel for the enterprise is the EDOC proposal [2], used for business process modelling.

   For serious enterprise design to be conducted, it is imperative that we have a means of relating the architectural description such other system views.

4. *Well-defined behavioural semantics to support formal analysis and verification.* A successful ADL for enterprise system modelling must (at least) meet the three challenges above. At the same time, we require that the ADL must retain a well-defined behavioral semantics. This is important, as this semantic aspect of an ADL description will benefit the enterprise developer, enabling formal analysis

and verification of the distributed aspects of the enterprise system.

The TrustME architectural description language (ADL) attempts to meet the challenges set above. In this paper, we focus on how the TrustME ADL enables description of *context-based interception* and *declarative deployment constraints* for middleware-based architectures.

For illustrative purposes, we outline how TrustME models a simple COM+-based architecture, now informally defined.

1.1. Example: Initial specification of problem Our example architecture involves a simple hotel reservation system, built from three COM+ components:

```
ReservationSystem
ReservationTransfer
ReservationBilling
```

Upon recieving an event notification from the first component that a hotel reservation is to be made by a user, the second component performes B2B operations with the hotel at which the reservation was made. Upon receiving the same type of event notification, the third component performes billing operations against the user's credit card. When the event is sent out, `ReservationTransfer` and `ReservationBilling` will execute concurrently. However, we require transaction support over both these components, as if one fails, then calls to either component must be rolled back. This will prevent a user being billed if the hotel they wish to book at is full, and will prevent the hotel from accepting a guest if their credit is bad.

We will describe this architecture, with a particular focus on how it represents a COM+ context-based interception transactional service.

2. The TrustME approach

The TrustME ADL was originally described in [7], and has been extended in [8] and [9]. TrustME is the product of continuing research conducted by the TrustME[1] group at the DSTC and DSSE. See [10] for relevant papers, including the most current version of this document.

TrustME decomposes a system into hierarchies of *kens*, linked to each other by connections between *gates*.

- *Kens* are self-contained, coarse-grain computational entities, potentially hierarchically composed from other kens. In general, we define a ken as a protection domain with well defined connections to and from other kens. If a ken contains other kens, then it is called *composite*, else it is called *primitive*.
- *Gates* protect kens, which cannot be directly accessed. All calls, all data communicated and all object migrated to a ken must come through gates. They are the ports for messages and migrating objects between kens. A direct connection from a ken's gate to another ken's gate designates a *use* relation.

Kens and gates are analogous to components and services respectively in Darwin, to components and ports respectively in C2 and ACME, or to processes and ports in MetaH [4].

However, TrustME differs from these other languages, in that it is augmented with further features to describe middleware-based applications.

2.1. Accomodating middleware component models Closer correspondence to middleware component models is achieved in several ways.

TrustME employs object-oriented mechanisms for its definitions. A ken is an instance of a *ken class*, and a gate is an instance of a *gate class*. These types of classes may be specialized through object-oriented subclassing. This permits reuse of specifications. Also, through subclassing, various aspects of system architecture can be modelling. Here, different types of software component (COM+ components, EJB components, etc) or componenent containers (COM+ servers, EJB container/servers, CORBA Orbs) can be modelled as different subclasses of ken. In this way, ken and gates have a broader range of uses than, say, components in Darwin.

For the purposes of enterprise computing modelling, the two important subclasses of kens are

- *Component kens*, used to represent components of a middleware-based system architecture. If a component ken provides a gate, this represents a component providing access to an interface. Similarly, a component's use of multiple interfaces is represented by the corresponding component ken providing a set of gates.

  A primitive component ken therefore corresponds to a black-box component (for example, a COTS component). A composite component ken corresponds to a grey-box component, in that some of the architectural details of the component's construction are visible, described by the ken's constituent subkens.

  The object-oriented nature of TrustME entails that component kens are closer to components of object-oriented middleware, because notions such as sub-

[1] Acronym for **Trust**ed Component **M**odel and Integration **E**nvironment for Distributed Services.

stitutability and subtyping are available to the designer.

- *Context kens* model contexts of a context-based interception system architecture, such as a COM+ server. A context ken always contains a set of *configuration attributes*.

  Configuration attributes are entities of TrustME that model contex settings for a context. Just as context settings in COM+ define how the context is to handle deployed components, a configuration attribute provides structural information about what services the context ken is meant to provide to deployed subkens. Semantically, a configuration attribute affects the dynamic behaviour of the subken interaction.

New ken subclasses can be devised to model other computational entities: for instance, a workflow engines, databases, or networks of machines.

Gates possess a richer language than, say, ports of Darwin.

- A gate is associated with an *interface* – that is, a signature for type checking.
- Gates are instances of gate classes. Consequently, gates are analogous to interface objects, adaptors or wrappers in programming.
- Designers can protect existing functionality within gates, but also it permits the substitutions between existing kens in a configuration, subject to compatibility checks over gates.

These features enable us retain a close correspondence with middleware coponent models. In particular, multiple interfaces for a middleware component are simply modelled by a component ken with multiple gates. This is difficult to achieve most other ADLs.

Example. (Cont.) The architecture of our example is represented diagrammatically in Fig. 1. The outermost ken is a context ken, `ReservationSystemContext`, with a set of configuration attributes. It contains three subkens, corresponding to the three components of the example. ⋄

2.2. Safer connections. The requirement for safer connections between components is also partly satified by assigning interfaces to gates, because validity of connections between components is determined by type checking.

Within the domain of programming language design, there are convincing arguments for adding further semantic annotations to component interfaces, to make interface usage safer. Essentially, it is argued that an interface signature alone does not tell us *what* the interface methods are supposed to do, and that this information is necessary to use the interface safely.

These arguments carry over to the ADL world. One popular means of providing semantic annotations for object-oriented designs is *design-by-contract* [11, 1]. In [7] and [9], the second author adapted design-by-contract to the architectural design level. Briefly,

- TrustME incorporates *rules* for defining how kens and gates should interact with each other. Rules provide TrustME with safer connections between components. Rules are public constraints, serving interface specification, according to the principles of design-by-contract.
- Rules can be associated with method interface specifications. Such rules specify properties of observable states before and after method execution (pre- and post- conditions). Rules can be associated with a particular gate. Such rules specify properties of observable states that must remain the same after any sequence of method executions (invariant conditions).
- Rules take the form of both logical predicates and finite state machines.

2.3. Integration with other software models, independent behavioural semantics We hope to achieve tighter integration with UML and MOF-based models, by defining TrustME as a metamodel within the MOF.

In particular, TrustME uses certain UML metamodel elements: Boolean propositions are given as OCL formulae, and finite state machines are taken as UML elements.

It is expected that a TrustME design can potentially build upon given UML software models. For instance, a component designed in UML might be represented as a primitive component ken in TrustME, but the state machines for the interfaces of the former may be used as rules for the gates of the latter.

An important feature of all ADLs is a rigorous behavioural semantics, to permit formal analysis of architectures. Although TrustME is a metamodel within the MOF, we also equip it with a separate nonvisual syntax and behavioural semantics. In this way, we achieve both conformance to standards and to formal analysis.

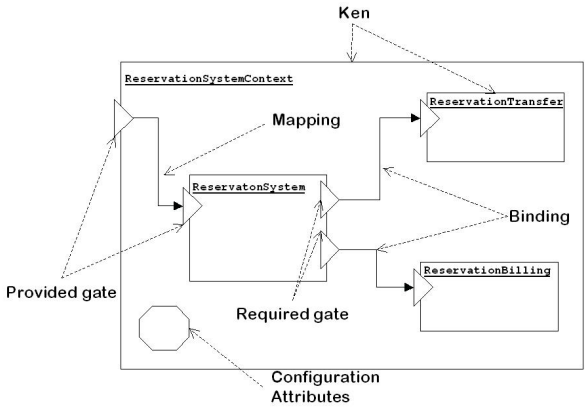

Figure 1: Representation of our example architecture, in terms of kens, gates, bindings, mappings and configuration attributes.

References

[1] OMG, "OMG Unified Modeling Language Specification," Tech. Rep., Object Management Group, 2000.

[2] OMG, "UML Profile for Enterprise Distributed Object Computing (EDOC) RFP," Tech. Rep., Object Management Group, 1999, Available at `ftp://ftp.omg.org/pub/docs/ad/99-03-10.pdf`.

[3] Alexander Ran, "Ares conceptual framework for software architecture," in *Software Architecture for Product Families: Principles and Practice*. 2000, pp. 1–29, Addison-Wesley.

[4] Nenad Medvidovic and Richard N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, Janurary 2000.

[5] Jeff Kramer, Jeff Magee, Keng Ng, and Naranker Dulay, "Software architecture description," in *Software Architecture for Product Families: Principles and Practice*. 2000, pp. 31–64, Addison-Wesley.

[6] OMG, "Meta Object Facility (MOF) Specification," Tech. Rep., Object Management Group, 2000.

[7] Heinz Schmidt, "Compatibility of interoperable objects," in *Information Systems Interoperability*. 1998, pp. 143–199, Research Studies Press, Taunton, Somerset, England.

[8] Sea Ling, Heinz Schmidt, and Rohan Fletcher, "Constructing interoperable components in distributed systems," in *IEEE Proceedings TOOLS Pacific '99, Melbourne*. 1999, pp. 274–284, IEEE Press.

[9] Heinz Schmidt and Ralf Reussner, "Automatic component adaptation by concurrent state machine retrofitting," Technical Report 2000/81, Monash University, School of Computer Science and Software Engineering, 2000.

[10] Iman Poernomo, Heinz Schmidt, and Ralf Reussner, "The TrustME language site," Web site, DSTC, 2001, Available at `http://www.csse.monash.edu.au/dsse/trustme`.

[11] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice/Hall, 1997.